C#Corner

.NET Core

Diving into .NET Core

Vinoth Arun Raj Xavier

Diving into Dot Net Core

Vinoth Arun Raj Xavier

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. Although the author/co-author and publisher have made every effort to ensure that the information in this book was correct at press time, the author/co-author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause. The resources in this book are provided for informational purposes only and should not be used to replace the specialized training and professional judgment of a health care or mental health care professional. Neither the author/co-author nor the publisher can be held responsible for the use of the information provided within this book. Please always consult a trained professional before making any decision regarding the treatment of yourself or others.

Author/Co-Author – Vinoth Arun Raj Xavier
Publisher – C# Corner
Editorial Team – Deepak Tewatia, Baibhav Kumar
Publishing Team – Praveen Kumar
Promotional & Media – Rohit Tomar, Rohit Sharma

About Author

I am Vinoth Arun Raj Xavier, with over 13 years of hands-on experience specializing in Microsoft.NET technology, various JavaScript frameworks, DevOps tools, and other essential programming languages.
My expertise lies in developing Windows, web, and mobile applications. I have a keen interest in microservices, cloud integration, and DevOps practices.
Continuously staying abreast of technological advancements across different technology stacks is something I find personally rewarding.
I focus on writing about current technical trends on social media platforms, sharing knowledge and insights with the community.

Table of Contents:

.NET Framework vs .NET Core 5

.NET Core Version History 8

Setting Up Your Development Environment 11

Essential Command Line Operations for Developers 24

Middleware and Key Components 36

Custom Middleware 28

What is Short-Circuiting? 33

1

.NET Framework vs .NET Core

Overview

This chapter provides an in-depth comparison between .NET Framework and .NET Core, highlighting their architecture, cross-platform compatibility, open-source nature, development models, application types, versioning, ecosystem, and migration pathways.

.NET Core: A Beginner's Guide to the Fundamentals

.NET Framework and .NET Core are both software development platforms created by Microsoft, but they have different goals, architecture, and use cases.

Here's a comparison of the two: -

Architecture

- **.NET Framework:** It is a Windows-only framework designed for building Windows desktop applications, web applications, and services. It has a monolithic architecture.
- **.NET Core:** It is designed as a cross-platform framework for building a variety of applications, including Windows, Linux, and macOS. It has a modular and lightweight architecture.

Cross-Platform Compatibility

- **.NET Framework:** Limited to Windows operating systems.
- **.NET Core:** Cross-platform, supporting Windows, Linux, and macOS.

Open Source

- **.NET Framework:** Closed source and primarily controlled by Microsoft.
- **.NET Core:** Open-source and developed in collaboration with the community. It serves as the foundation for .NET 5 and later.

Development Model

- **.NET Framework:** Uses the full .NET Framework libraries. Development typically involves Visual Studio.
- **.NET Core:** Uses the .NET Core libraries. Development can be done using Visual Studio or Visual Studio Code, as well as on other platforms with various IDEs.

Application Types

- **.NET Framework:** Primarily used for Windows desktop applications, ASP.NET web applications, and Windows services.
- **.NET Core:** Supports a wider range of application types, including console applications, web applications (ASP.NET Core), and microservices.

Versioning

- **.NET Framework:** Released in specific versions (e.g., 4.0, 4.5, 4.8).
- **.NET Core:** Released in a more predictable and frequent manner, with version numbers like .NET Core 1.x, 2.x, and .NET 5, 6, etc.

Ecosystem

- **.NET Framework:** Has a mature and established ecosystem with many libraries and third-party components.
- **.NET Core:** While it had a growing ecosystem, it merged into .NET 5 and later, which includes a broader range of libraries and components.

Migration

- **.NET Framework:** Migrating legacy applications to .NET Core or .NET 6+ may require significant effort due to architectural and API differences.
- **.NET Core:** Applications built with .NET Core are easier to migrate to .NET 5+ since .NET Core is part of the .NET 5+ ecosystem.

In summary, if you are starting a new project or planning to port existing applications to a modern, cross-platform environment, .NET 6 (or later) or its successors under the ".NET" umbrella is the way to go. However, for maintaining legacy Windows applications, .NET Framework is still a valid choice.

2

.NET Core Version History

Overview

In this chapter, we will see the history of the .NET Core versions.

Here is a list of the major .NET Core versions, along with some key details about each version:

.NET Core 1.0 (Initial Release)

- **Released:** June 2016
- First official release of .NET Core.
- Supported cross-platform development but was relatively limited in features compared to later versions.

.NET Core 1.1

- **Released:** November 2016
- Focused on performance improvements and bug fixes.
- Maintained compatibility with .NET Standard 1.6.

.NET Core 2.0

- **Released:** August 2017
- Introduced many new features and improvements.
- Expanded API coverage and added support for .NET Standard 2.0.

.NET Core 2.1

- **Released:** May 2018
- Focused on performance enhancements and improving existing features.
- Long-Term Support (LTS) release.

.NET Core 2.2

- **Released:** December 2018
- Included some new features and updates.
- Also, an LTS release, with support until December 2019.

.NET Core 3.0

- **Released:** September 2019
- Introduced significant improvements, including enhanced support for Windows Desktop applications.
- Expanded platform support and the introduction of .NET Core 3.0 SDK.

.NET Core 3.1

- **Released:** December 2019
- Focused on performance improvements and bug fixes.
- Long-Term Support (LTS) release with support until December 2022.

.NET 5

- **Released:** November 2020
- A major shift in branding, as it was the first version released under the ".NET" umbrella, merging .NET Core and .NET Framework features.
- Not an LTS release, but it set the stage for future .NET releases.

.NET 6

- **Released:** November 2021
- A significant release that aimed to unify .NET development for various workloads.

- Focused on enhancing performance, improving cloud-native capabilities, and expanding platform support.
- An LTS release with support until November 2024.

.NET 7

- **Released:** November 2022
- Focused on Servers and Runtime, Minimal API, and Middleware Updates.
- Breaking Changes
- An STS release with support until May 14, 2024.

.NET 8

- Announcing .NET 8 Release Candidate 2 - .NET Blog (microsoft.com)

3

Setting Up Your Development Environment

Overview

In this chapter, we will see the configuration of a .NET Core development environment on Windows, Linux, and Mac platforms.

Configuring a .NET Core development environment on Windows, Linux, and Mac involves a series of steps to ensure you have the necessary tools and components.

Here's a general guide for each platform:

Windows

1. **Install Visual Studio:** Download and install Visual Studio, which is the official integrated development environment (IDE) for .NET development on Windows. Make sure to select the ".NET Core" workload during installation.
2. **Install .NET Core SDK:** Download and install the .NET Core SDK for Windows. You can find the installer on the official .NET website. This SDK includes the necessary tools to build and run .NET Core applications.
3. **Set up an Editor:** While Visual Studio is the primary choice, you can also use Visual Studio Code, a lightweight code editor, for .NET Core development. Install Visual Studio Code and add relevant extensions for .NET Core.

Linux

1. **Install .NET Core SDK:** On Linux, you can install the .NET Core SDK using package managers like APT or YUM or by downloading the official installer from the .NET website. The installation method depends on your specific Linux distribution.
2. **Set Up an Editor:** You can use a code editor of your choice, such as Visual Studio Code or JetBrains Rider, for .NET Core development. Install the editor and the necessary extensions.

Mac

1. **Install .NET Core SDK:** On a Mac, you can download and install the .NET Core SDK directly from the official .NET website.
2. **Set Up an Editor:** Like Linux, you can choose your preferred code editor, like Visual Studio Code or JetBrains Rider, for .NET Core development. Install the editor and relevant extensions.
3. **Optional: Homebrew (macOS):** If you're using macOS, you can use Homebrew to install the .NET Core SDK. Install Homebrew if you haven't already, and then run the command brew install --cask dotnet-sdk to install the .NET Core SDK.
4. **Verify Installation:** After installation, open a terminal and run the dotnet --version to ensure that the .NET Core SDK is correctly installed.
5. **Create and Build Projects:** You can now create and build .NET Core projects using the dotnet new and dotnet build commands.

Remember to refer to the official documentation and installation guides provided by Microsoft for the most up-to-date instructions. Additionally, it's essential to keep your development environment updated with the latest SDK and tooling versions to take advantage of new features and improvements

4

Essential Command Line Operations for Developers

Overview

In this chapter, we will see .NET Core command-line commands, including project creation, dependency management, building, running, publishing, and testing. Additionally, it covers creating solutions, managing Entity Framework migrations, and working with worker services.

Here are some useful commands for working with .NET Core from the command line.

Creating a New .NET Core Project

- To create a new console application: dotnet new console -n YourProjectName
- To create a new ASP.NET Core web application: dotnet new web -n YourWebProjectName

Restoring Dependencies

- To restore project dependencies (e.g., NuGet packages): dotnet restore

Building a .NET Core Project

- To build a project: dotnet build

Running a .NET Core Application

- To run the application: dotnet run

Creating a New Solution

- To create a new solution: dotnet new sln -n YourSolutionName

Adding a Project to a Solution

- To add a project to a solution: dotnet sln YourSolutionName.sln add YourProjectName/Path.csproj

Publishing an Application

- To publish a self-contained application (including runtime): dotnet publish -c Release -r win-x64

Managing NuGet Packages

- To add a NuGet package to a project: dotnet add package PackageName
- To remove a NuGet package from a project: dotnet remove package PackageName

Managing Entity Framework Migrations (for Database)

- To create a new migration: dotnet ef migrations add MigrationName
- To apply pending migrations to the database: dotnet ef database update

Running Tests

- To run unit tests using xUnit or other testing frameworks: the dotnet test

Publishing as a Single Executable (with .NET 6 and later)

- To publish as a single executable: dotnet publish -r os-x64 -p:PublishSingleFile=true

Creating a New Worker Service

- To create a new worker service: dotnet new worker -n YourWorkerServiceName

Managing Entity Framework Database Context

- To generate code for the DbContext and entities from the database: dotnet ef dbcontext scaffold "Connection String" Microsoft.EntityFrameworkCore.SqlServer -o OutputFolder

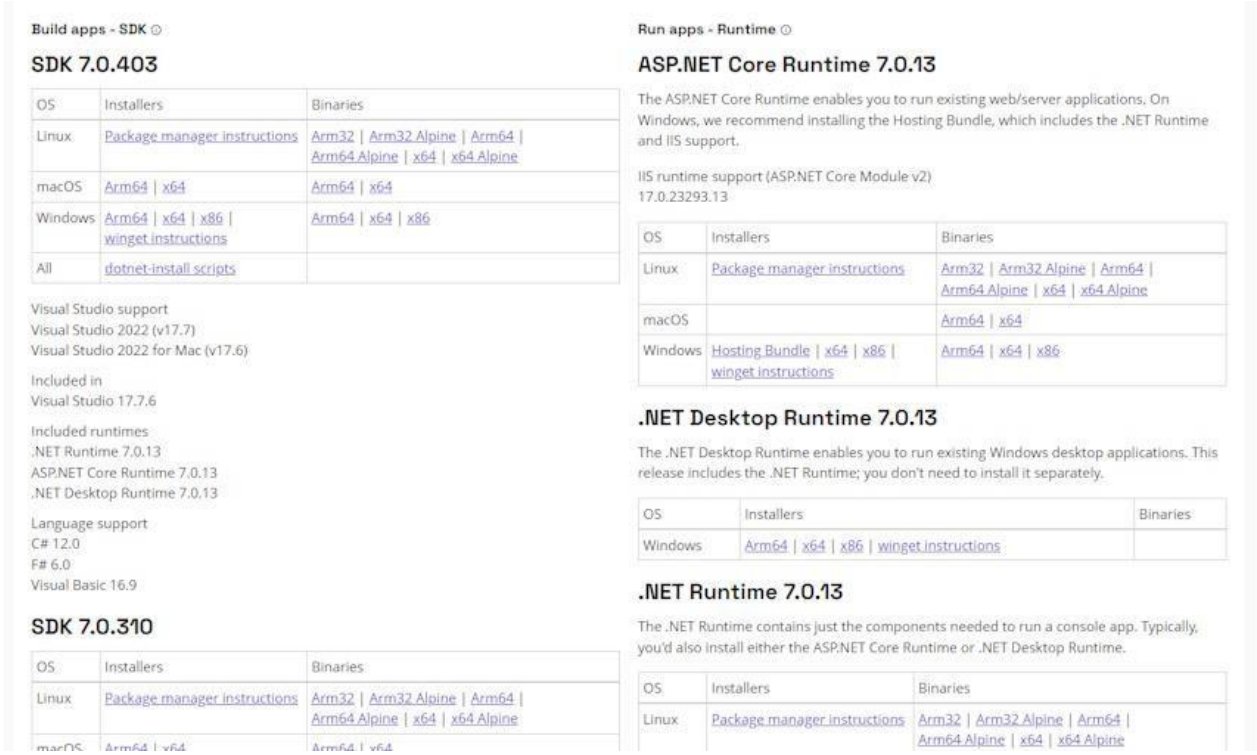
These are some of the fundamental commands for working with .NET Core. Depending on your project and requirements, you may use additional commands and options. You can always get more information about a specific command by running dotnet command --help, such as dotnet build --help or dotnet run --help, to see the available options and parameters for that command.

.NET SDK

We would use the .NET SDK to develop applications. This SDK includes everything you need to get started with .NET Core development, from creating new projects to deploying your applications. Visual Studio and Visual Studio Code are popular Integrated Development Environments (IDEs) that seamlessly integrate with the .NET SDK, making it easier for developers to write, test, and debug their code. Both IDEs offer features like code completion, debugging tools, and project management capabilities to enhance the development experience.
By using the .NET SDK in conjunction with Visual Studio or Visual Studio Code, developers can efficiently create a wide range of applications, from web and desktop applications to cloud services and more. These tools empower developers to bring their ideas to life and streamline the development process, ultimately leading to more robust and scalable software solutions.

.NET Core Runtime

The ASP.NET Core Runtime is a component of the ASP.NET Core framework that's responsible for executing and running ASP.NET Core applications. It's a lightweight and essential part of the framework that provides the necessary environment and libraries to host and run web applications built using ASP.NET Core.
In a nutshell, the ASP.NET Core Runtime is like the engine that powers ASP.NET Core web applications. It handles tasks such as processing HTTP requests, managing routing, and interacting with the web server. It's crucial for deploying and hosting ASP.NET Core applications on a web server, making them accessible to users over the internet.



Build apps - SDK

SDK 7.0.403

OS	Installers	Binaries
Linux	Package manager instructions	Arm32 | Arm32 Alpine | Arm64 | Arm64 Alpine | x64 | x64 Alpine
macOS	Arm64 | x64	Arm64 | x64
Windows	Arm64 | x64 | x86 | winget instructions	Arm64 | x64 | x86
All	dotnet-install scripts	

Visual Studio support
Visual Studio 2022 (v17.7)
Visual Studio 2022 for Mac (v17.6)

Included in
Visual Studio 17.7.6

Included runtimes
.NET Runtime 7.0.13
ASP.NET Core Runtime 7.0.13
.NET Desktop Runtime 7.0.13

Language support
C# 12.0
F# 6.0
Visual Basic 16.9

SDK 7.0.310

OS	Installers	Binaries
Linux	Package manager instructions	Arm32 | Arm32 Alpine | Arm64 | Arm64 Alpine | x64 | x64 Alpine
macOS	Arm64 | x64	Arm64 | x64

Run apps - Runtime

ASP.NET Core Runtime 7.0.13

The ASP.NET Core Runtime enables you to run existing web/server applications. On Windows, we recommend installing the Hosting Bundle, which includes the .NET Runtime and IIS support.

IIS runtime support (ASP.NET Core Module v2)
17.0.23293.13

OS	Installers	Binaries
Linux	Package manager instructions	Arm32 | Arm32 Alpine | Arm64 | Arm64 Alpine | x64 | x64 Alpine
macOS		Arm64 | x64
Windows	Hosting Bundle | x64 | x86 | winget instructions	Arm64 | x64 | x86

.NET Desktop Runtime 7.0.13

The .NET Desktop Runtime enables you to run existing Windows desktop applications. This release includes the .NET Runtime; you don't need to install it separately.

OS	Installers	Binaries
Windows	Arm64 | x64 | x86 | winget instructions	

.NET Runtime 7.0.13

The .NET Runtime contains just the components needed to run a console app. Typically, you'd also install either the ASP.NET Core Runtime or .NET Desktop Runtime.

OS	Installers	Binaries
Linux	Package manager instructions	Arm32 | Arm32 Alpine | Arm64 | Arm64 Alpine | x64 | x64 Alpine

.NET Core Projects

Please check the latest .NET documentation for any updates or additional project types.

1. **Console Application:** Used for building command-line tools and utilities. These applications run in a console window and are ideal for automation scripts.
2. **Class Library:** A library project that contains reusable classes, methods, and components to be used in other .NET projects.
3. **ASP.NET** Core Web Application
    a) **MVC:** Used for building web applications following the Model-View-Controller (MVC) architectural pattern.
    b) **API:** Used for creating RESTful APIs.
    c) **Razor Pages:** A simplified web application model for building page-based web applications.
4. **Blazor Web Application:** Allows you to build web applications using Blazor, which enables C# code to run directly in the browser.
5. **SignalR Hub:** Used for building real-time, bi-directional communication between clients andthe server.
6. **Worker Service:** Designed for background services or long-running processes that do not require user interaction, often used for tasks like batch processing or data synchronization.
7. **Azure Functions:** A serverless computer service that lets you run event-triggered code without having to manage infrastructure.
8. **Xamarin.Forms App (Mobile):** Used for cross-platform mobile app development, where you can write a single codebase for iOS, Android, and Windows using Xamarin.
9. **Windows Forms App (Desktop):** Enables the creation of traditional Windows desktop applications with a graphical user interface.
10. **WPF App (Desktop):** Allows you to build Windows Presentation Foundation (WPF) applications, which are used for creating rich desktop applications.
11. **Blazor Component Library**: A project type for building reusable Blazor components that can be shared across multiple Blazor applications.
12. **Entity Framework Core Project:** A project used for data access and database interactions. It can be used with ASP.NET Core applications to manage data models.
13. **NUnit, xUnit, or MSTest Projects:** These are used for creating unit tests to ensure the correctness and reliability of your code.
14. **Docker Support:** Some project templates come with Docker support, allowing you to containerize your application for easier deployment and scalability.

Please note that .NET and its project templates may have evolved further since my last update. I recommend visiting the official .NET documentation or using the latest version of Visual Studio to explore the available project templates and their capabilities.

Exploring Dependency Injection

Understanding and using the built-in dependency injection system in .NET Core is a crucial aspect of modern application development. Dependency injection is a design pattern that helps manage object dependencies and promotes loose coupling between components, making your code more

maintainable and testable. Here's how to understand and use the built-in dependency injection system in .NET Core:

- **Add the Required NuGet Packages**: Ensure that you have the necessary NuGet packages installed. In .NET Core, the primary package is Microsoft.Extensions.DependencyInjection. You can add this package to your project using the package manager console or Visual Studio's NuGet Package Manager.
- **Service Registration:** Dependency injection in .NET Core revolves around registering and resolving services. Services are objects that provide certain functionality to your application. You need to register your services in the application's startup code, typically in the Startup.cs file.

Understanding and using the built-in dependency injection system in .NET Core is a crucial aspect of modern application development. Dependency injection is a design pattern that helps manage object dependencies and promotes loose coupling between components, making your code more maintainable and testable. Here's how to understand and use the built-in dependency injection system in .NET Core:

- **Add the Required NuGet Packages**: Ensure that you have the necessary NuGet packages installed. In .NET Core, the primary package is Microsoft.Extensions.DependencyInjection. You can add this package to your project using the package manager console or Visual Studio's NuGet Package Manager.
- **Service Registration:** Dependency injection in .NET Core revolves around registering and resolving services. Services are objects that provide certain functionality to your application. You need to register your services in the application's startup code, typically in the Startup.cs file.

```
public void ConfigureServices(IServiceCollection services)
{
// Register services here
services.AddScoped<IUser, InternalUser>();
}
```

  - **AddScoped:** The service is created once per client request (scoped to the request's lifetime).
  - **AddTransient:** The service is created each time it's requested.
  - **AddSingleton:** The service is created once for the entire application lifetime.
- **Inject Services into Your Classes**: After registering your services, you can inject them into your classes that depend on these services. This is typically done through constructor injection. For example:
- **Dependency Injection Container:** The.ConfigureServices method you see in the Startup class is responsible for building and configuring the dependency injection container. This container manages the lifetime and creation of registered services.
- **Using the Services:** You can now use the injected services within your classes. For example, in a controller, you can invoke methods on the injected service.

```
public async Task<Result> GetUsers()
{
    var result = await _user.GetAllAsync();
    // ...
}
```

- **Scopes and Lifetime Management:** Understand the different lifetimes (Scoped, Transient, and Singleton) and choose the appropriate one for your services. This ensures that services behave as expected and are created and disposed of correctly.
- **Testing and Mocking:** One of the key benefits of dependency injection is that it simplifies unit testing. You can easily replace actual services with mock objects in your tests, allowing you to isolate and test your code's behavior.
- **Advanced Usage**: .NET Core's dependency injection system also supports more advanced scenarios, like named instances, options configuration, and custom lifetimes

The built-in dependency injection system in .NET Core simplifies managing object dependencies, promotes better code organization, and facilitates unit testing. It's a fundamental concept for building robust and maintainable applications in the .NET ecosystem.

Scopes and Lifetime Management

In .NET Core's built-in dependency injection (DI) system, the concept of "scopes and lifetime management" refers to how long instances of registered services are kept alive and how they are shared within an application. Understanding the different lifetimes available for services is essential for managing resources, performance, and the correct behavior of your application. Here are the details of the three main lifetimes available:

Transient

- Transient services are created every time they are requested.
- A new instance of a transient service is created for each request or injection.
- They have the shortest lifespan, suitable for stateless and lightweight services.

```
services.AddTransient<IService, Service>();
```

Scoped

- Scoped services are created once per client request (HTTP request in the case of web applications).
- Within the same HTTP request, all injections of a scoped service share the same instance.
- This is useful for maintaining the state during a single request, such as a database context.

```
services.AddScoped<IService, Service>();
```

Singleton

- Singleton services are created once for the entire application lifetime.
- All injections of a singleton service share the same instance.
- They are suitable for stateful services that should be shared across the entire application.

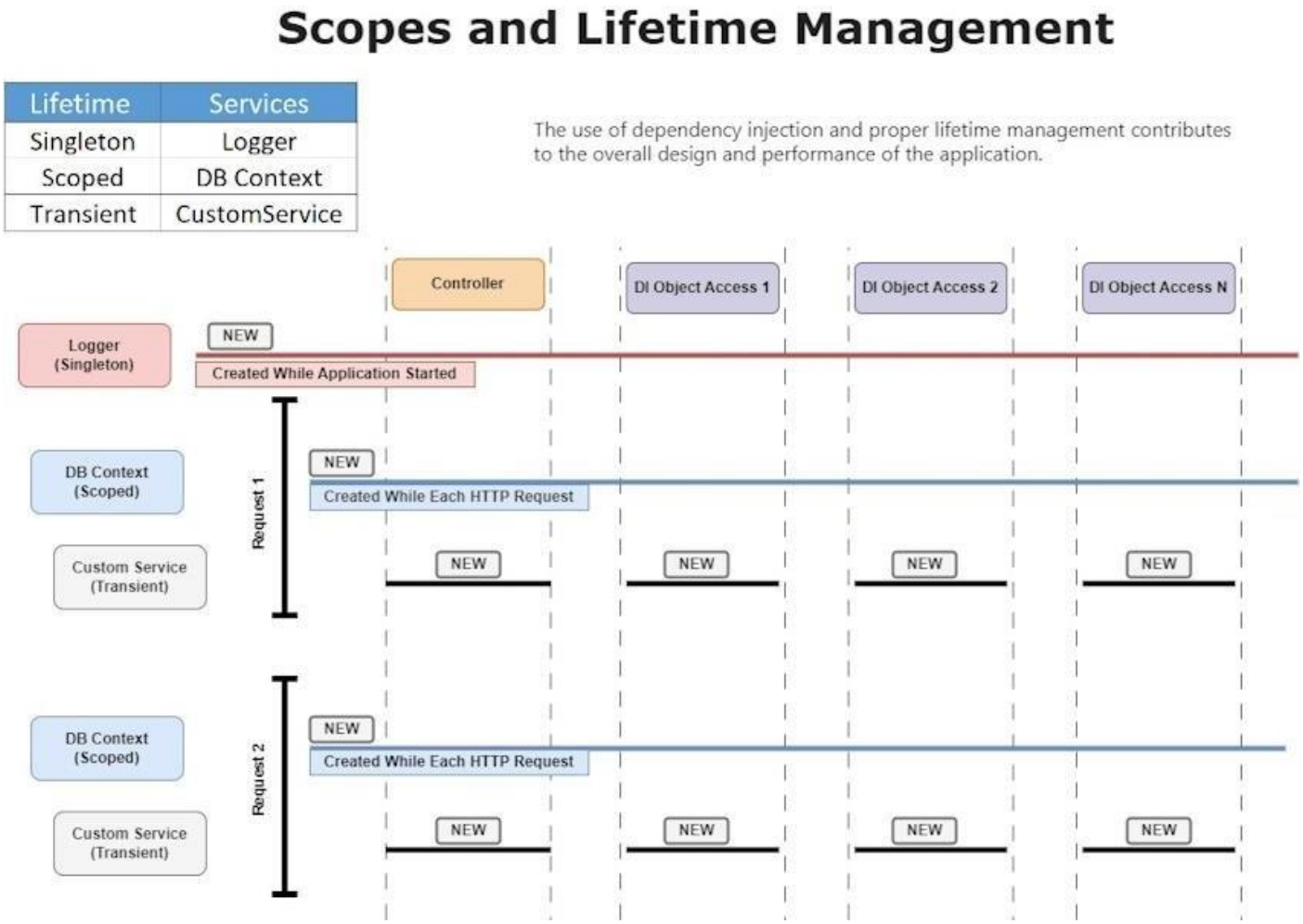
```
services.AddSingleton<ILogService, LogService>();
```

You register services with their respective lifetimes in the ConfigureServices method of the Startup class, as shown in the examples above.

Short Note

- **Transient:** Use for lightweight, stateless services that don't maintain any internal state. Each usage results in a new instance.
- **Scoped:** Ideal for services that should share data and state within a single client request. Commonly used for database contexts and unit-of-work patterns.
- **Singleton:** Suitable for services that maintain a shared state across the entire application. Be cautious with mutable states, as changes may affect other parts of your application.

Simple Image Illustration.



Interface like below

```
namespace scopesandlifetimeservices.Services
{
    public interface IScopedService : IDisposable
    {
        Guid ScopeId { get; set; }
    }

    public interface ISingletonService : IDisposable
    {
        Guid SingletonId { get; set; }
    }

    public interface ITransientService : IDisposable
    {
        Guid TransientId { get; set; }
    }
}
```

Implement the interfaces

```
namespace scopesandlifetimeservices.Services
{
    public class ScopedService : IScopedService
    {
        public Guid ScopeId { get; set; }

        public void Dispose() => GC.Collect();

        public ScopedService()
        {
            ScopeId = Guid.NewGuid();
        }
    }

    public class SingletonService : ISingletonService
    {
        public Guid SingletonId { get; set; }

        public void Dispose() => GC.Collect();

        public SingletonService()
        {
            SingletonId = Guid.NewGuid();
        }
    }
```

```
    public class TransientService : ITransientService
    {
        public Guid TransientId { get; set; }

        public void Dispose() => GC.Collect();

        public TransientService()
        {
            TransientId = Guid.NewGuid();
        }
    }
}
```

In Controller

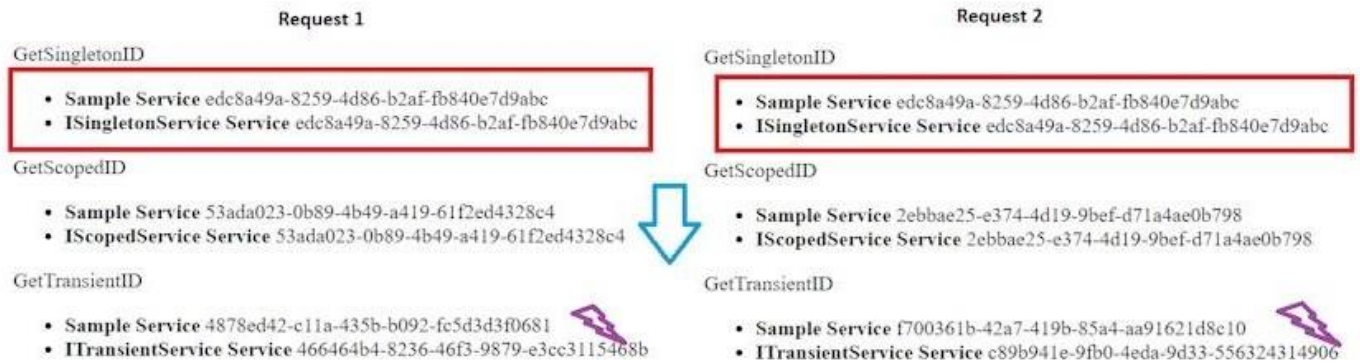
```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using scopesandlifetimeservices.Services;

namespace scopesandlifetimeservices.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class LifeTimeController : ControllerBase
    {
        readonly ISingletonService _singletonService;
        readonly IScopedService _scopedService;
        readonly ITransientService _transientService;
        readonly ISample _sample;

        public LifeTimeController(
            ISingletonService singletonService,
            IScopedService scopedService,
            ITransientService transientService,
            ISample sample)
        {
            _singletonService = singletonService;
            _scopedService = scopedService;
            _transientService = transientService;
            _sample = sample;
        }

        [HttpGet("request")]
        public ActionResult RequestMade()
        {
            var message = $"<!DOCTYPE html><html lang='en'><meta
charset='UTF-8'><meta " +
```

```
                $"content='width=device-width, initial-scale=1'
name='viewport'><title>Scopes and Lifetime " +
                $"Management</title><h1>Scopes and Lifetime
Management.</h1>" +
                $"GetSingletonID<ul><li><b>Sample Service " +
                $"</b>{_sample.GetSingletonID()}<li><b>ISingletonService
Service</b> " +
                $"{_singletonService.SingletonId}</ul>" +
                $"GetScopedID<ul><li><b>Sample Service " +
                $"</b>{_sample.GetScopedID()}<li><b>IScopedService Service
" +
                $"</b>{_scopedService.ScopeId}</ul>" +
                $"GetTransientID<ul><li><b>Sample Service " +
                $"</b>{_sample.GetTransientID()}<li><b>ITransientService
Service</b> " +
                $"{_transientService.TransientId}</ul>";

            return new ContentResult
            {
                Content = message, ContentType = "text/html"
            };
        }
    }
}
```

In the Startup of the program

```
builder.Services.AddSingleton<ISingletonService,SingletonService>();
builder.Services.AddScoped<IScopedService, ScopedService>();
builder.Services.AddTransient<ITransientService, TransientService>();
builder.Services.AddScoped<ISample, SampleDIConsumer>();
```

Response

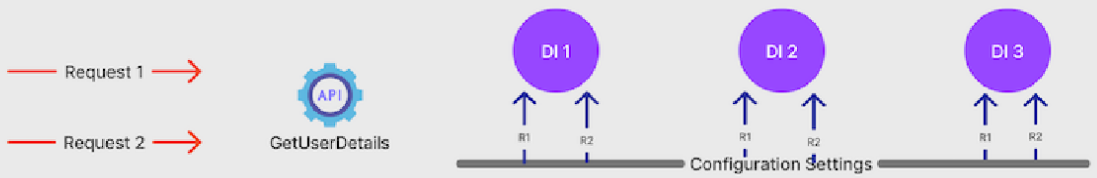


Scopes and Lifetime Management

Type	Services
Singleton	Configuration Settings
Scoped	DBContext
Transient	CustomService1

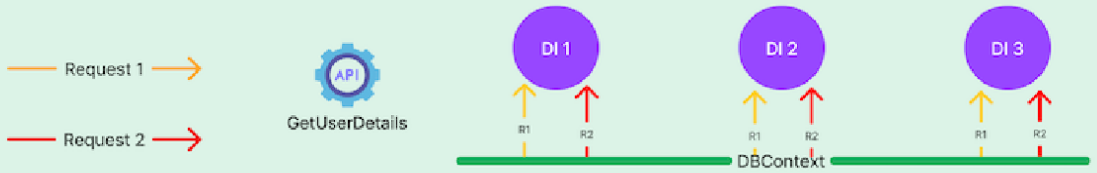
Singleton

Suitable for services that maintain a **shared state across the entire application**. Be cautious with mutable states, as changes may affect other parts of your application.



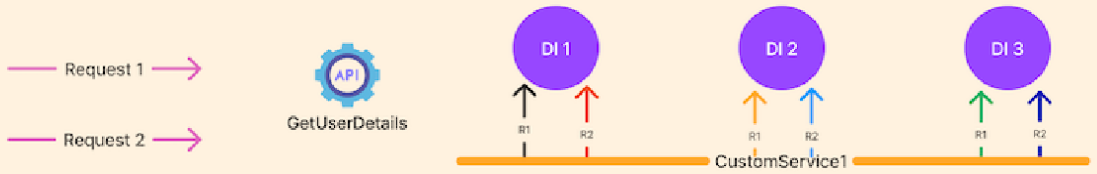
Scoped

Ideal for services that should **share data and state within a single client request**. Commonly used for database contexts and unit-of-work patterns.



Transient

Use for lightweight, stateless services that don't maintain any internal state. **Each usage results in a new instance.**



Sample Request

Request 1

GetSingletonID

- **Sample Service** 9e0d6bb7-eb8a-40fb-ab07-a4cef1d799d3
- **ISingletonService Service** 9e0d6bb7-eb8a-40fb-ab07-a4cef1d799d3

GetScopedID

- **Sample Service** 7c93e7bb-2c5e-4e84-acf8-553a7582e0cf
- **IScopedService Service** 7c93e7bb-2c5e-4e84-acf8-553a7582e0cf

Request 2

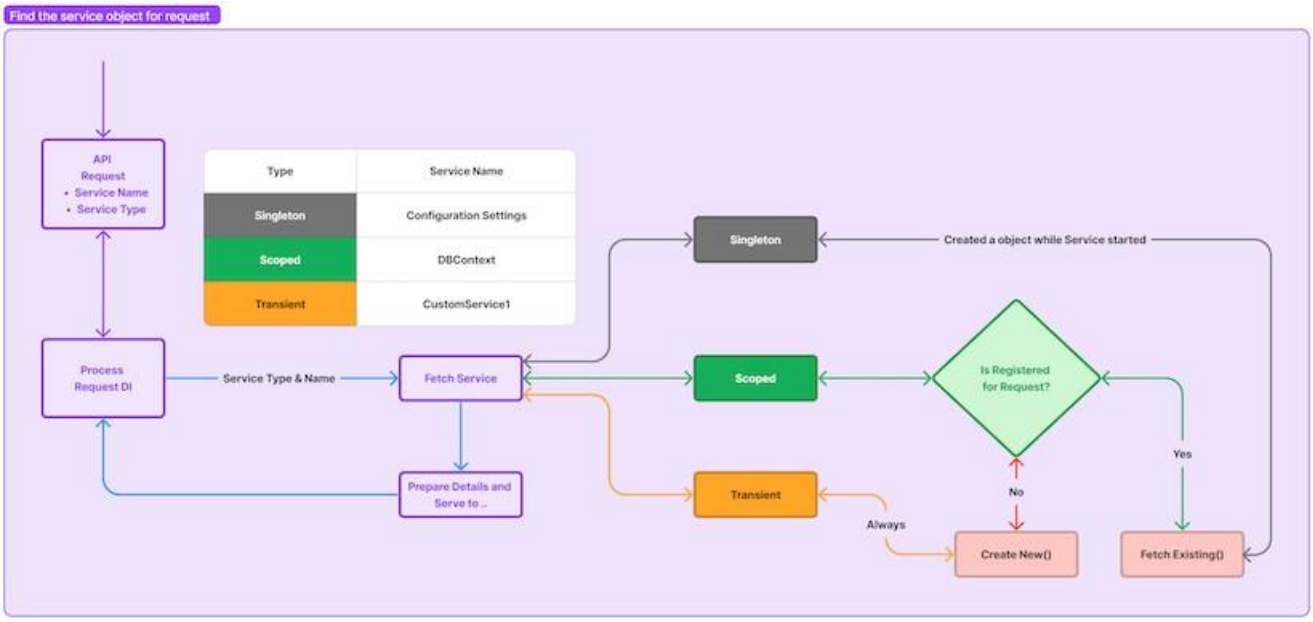
GetSingletonID

- **Sample Service** 9e0d6bb7-eb8a-40fb-ab07-a4cef1d799d3
- **ISingletonService Service** 9e0d6bb7-eb8a-40fb-ab07-a4cef1d799d3

GetScopedID

- **Sample Service** c3f44aef-1df8-4ed3-a646-550939ef6264
- **IScopedService Service** c3f44aef-1df8-4ed3-a646-550939ef6264

The below image will explain the flow of getting services from .net core service registry,



Once a request comes to the endpoint,

- The process begins by fetching the required service details from the service registry, which involves specifying the service type and service name. For example, the service type is "Singleton," and the service name is "IConfigurationSettings" from the Dependency Injection (DI) of the Controller.
- The system then checks the service type:
    - If it's "Singleton," it does not create a new object. Instead, it retrieves it from the already created object and returns it. This object is shared memory throughout the application.
    - If it's "Scoped," it checks whether the request already has shared memory within the request. If so, it returns the existing object; otherwise, it creates a new object and returns it. This shared memory is specific to the single request of the endpoint.
    - If it's "Transient," it does not maintain shared memory, always creating a new object and returning it.
- In our example, since it's Singleton, the system retrieves the already created object of settings from the application's startup.

5

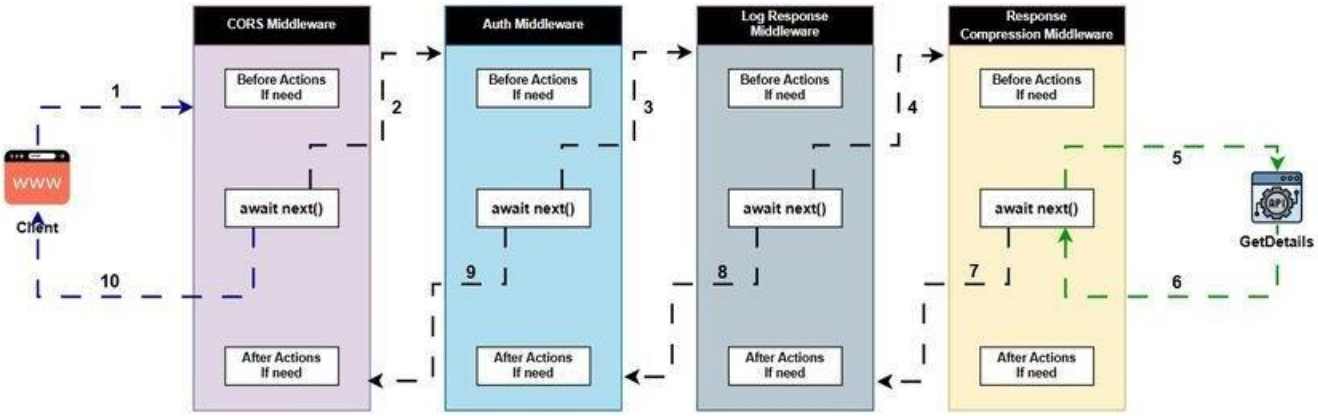
Middleware and Key Components

Overview

In this chapter, we will explore .NET Core middleware, explaining its role in request processing pipelines. It covers components like authentication, authorization, logging, error handling, and CORS, highlighting their importance and providing insights into configuration methods and short-circuiting.

What is Middleware?

Middleware is integrated into an application pipeline to oversee the handling of requests and responses. Each component in the pipeline decides whether to pass the request to the next one. It has the capability to function both before and after the subsequent pipeline component. The request pipeline is assembled using requested delegates, and each HTTP request is managed by these delegates.
In configuring request delegates, the Run, Map, and Use extension methods play a key role. A request delegate, which serves as an individual processing step, can be set up either in-line as an anonymous method (referred to as in-line middleware) or within a reusable class. These classes and in-line methods collectively form middleware components. Each middleware component within the request pipeline has the responsibility of either invoking the next component in the sequence or, in certain cases, halting the pipeline. When a middleware component halts the process, it earns the term "terminal middleware" as it effectively prevents subsequent middleware from further handling the request.



Understanding Middleware: Middleware, in the context of .NET Core, refers to software components that are assembled into the request processing pipeline to handle requests and responses. These components execute in a sequential order, providing a flexible mechanism to perform tasks such as authentication, authorization, logging, error handling, and more. Each middleware component has a specific responsibility and can modify the request or response as it passes through the pipeline.

Key Components and Functionality

- **Authentication Middleware:** One of the primary functions of middleware is handling authentication. .NET Core middleware allows developers to integrate authentication mechanisms seamlessly into the pipeline. This is crucial for ensuring that only authorized users can access protected resources.
- **Authorization Middleware:** Authorization middleware complements authentication by determining whether an authenticated user has the necessary permissions to perform a specific action or access certain resources. This layer adds an additional level of security to the application.

- **Logging Middleware:** Middleware can be employed for logging purposes, capturing valuable information about each request and response. This aids in diagnosing issues, monitoring application performance, and analyzing user behavior.
- **Exception Handling Middleware:** Errors and exceptions are inevitable in any application. Middleware facilitates the centralized handling of exceptions, ensuring that developers can implement custom error pages, log detailed error information, and maintain a consistent user experience.
- **CORS Middleware:** Cross-Origin Resource Sharing (CORS) is a security feature implemented by middleware to control which domains can access resources on a server. This is crucial for securing web applications and preventing unauthorized access.
- **Compression Middleware:** To enhance performance, middleware can be used to compress responses before they are sent to clients. This reduces the amount of data transferred over the network, resulting in faster load times for users.

Advantages of .NET Core Middleware

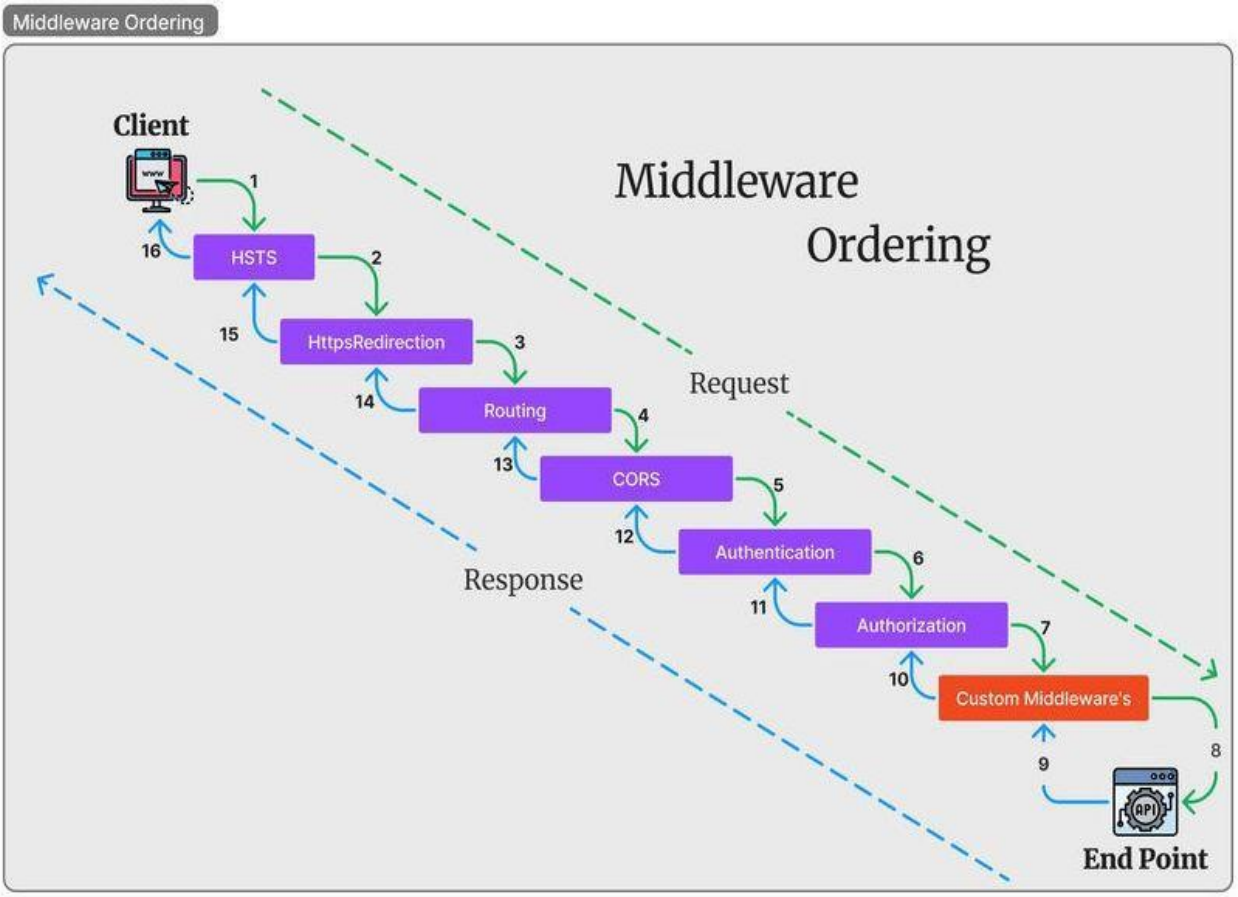
- **Modularity and Reusability:** Middleware promotes modularity by allowing developers to encapsulate specific functionality into reusable components. This modular approach simplifies code maintenance and promotes the reuse of middleware across different projects.
- **Pipeline Customization:** Developers have the flexibility to customize the request-response pipeline based on the specific requirements of their application. This adaptability is crucial for tailoring the application's behavior to meet unique business needs.
- **Improved Testability:** Middleware components can be unit-tested in isolation, promoting better testability, and facilitating the adoption of test-driven development (TDD) practices. This ensures that each middleware component functions correctly and adheres to its intended behavior.
- **Centralized Configuration:** .NET Core middleware allows for centralized configuration, making it easier to manage and modify the behavior of various components. This simplifies the task of adjusting application settings without the need to modify code.

Is Middleware Ordering Important?

Yes, in ASP.NET Core, the order in which middleware components are added to the pipeline is significant. Middleware components are executed in the order they are added to the pipeline. This means that the order of middleware registration affects the order of their execution during the request processing pipeline.

The order is important because each middleware component might modify the request or response, and the order in which they are executed can impact the final outcome. For example, if you have authentication middleware, it should be executed before other middleware that relies on authenticated user information.

Keep in mind that some middleware components are order-sensitive, while others may not be. Always check the documentation for specific middleware to understand any ordering requirements or recommendations.

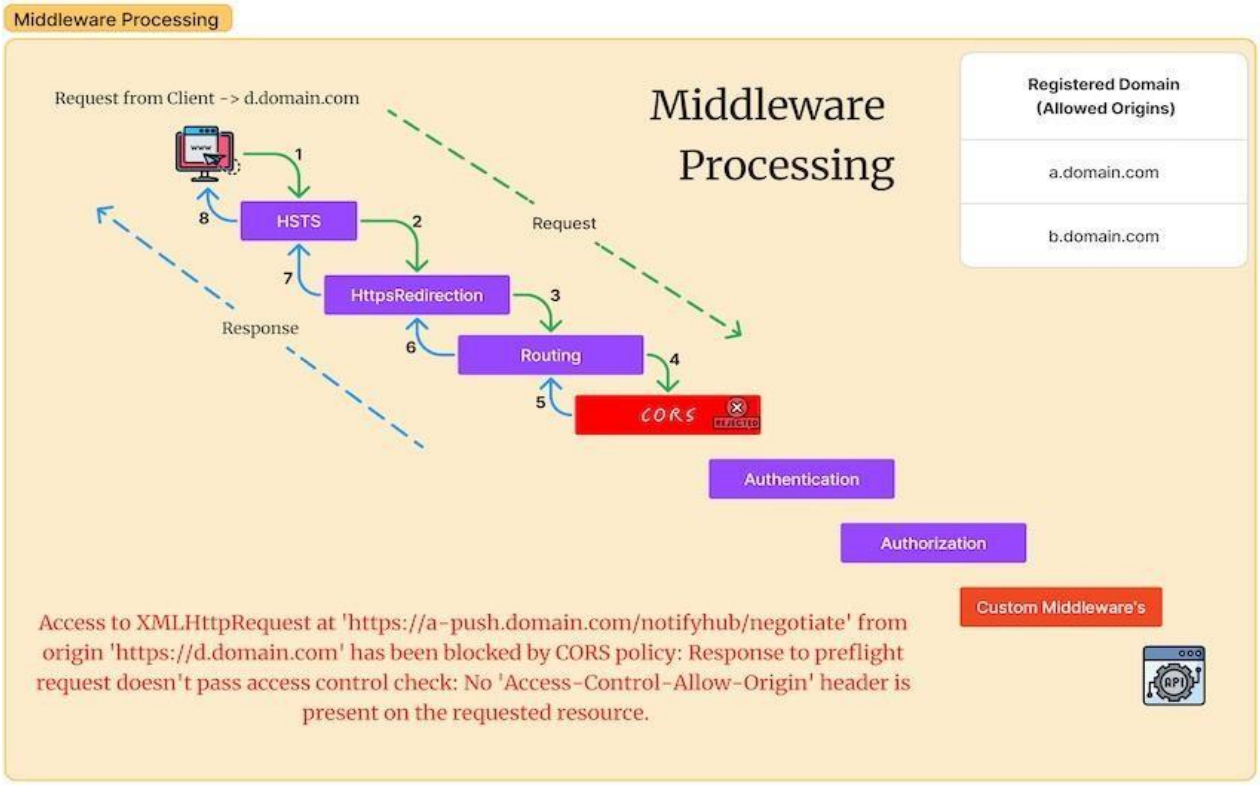


Referred in this article.

Actual Process of Middleware

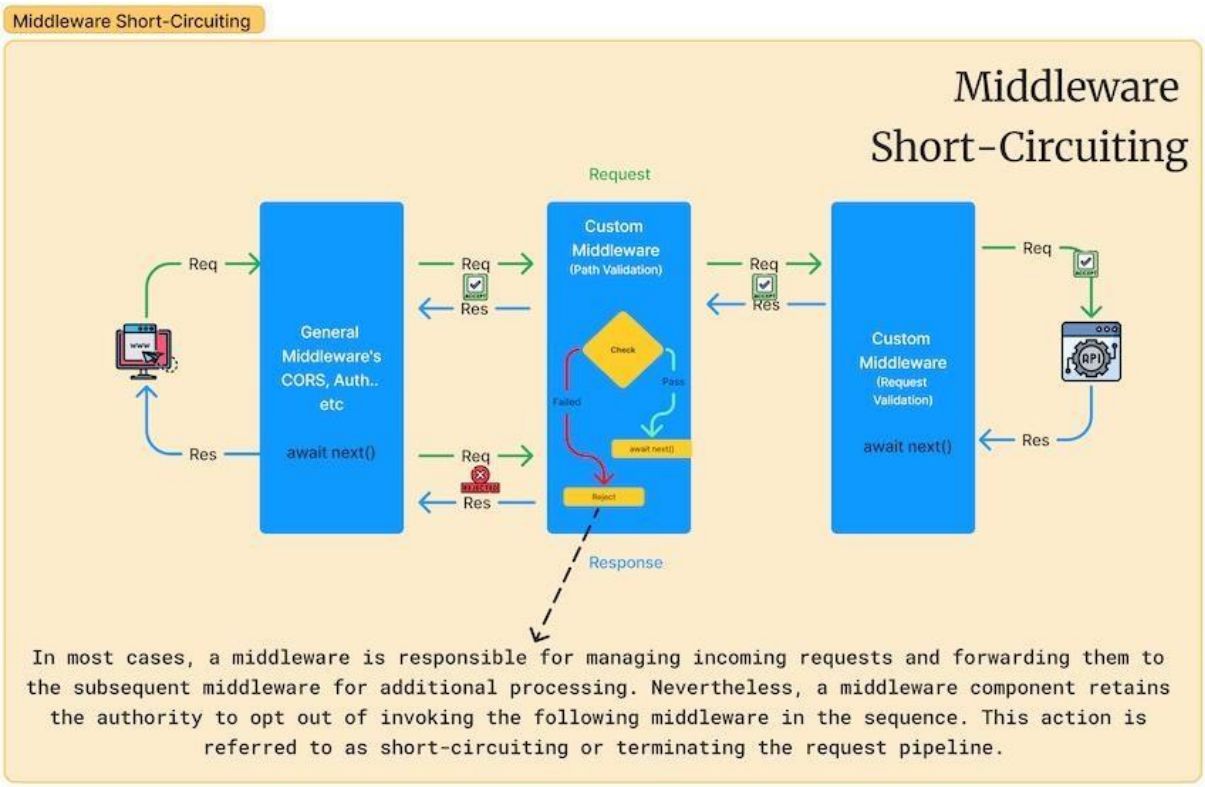
In the given scenario, a client-initiated a request from the 'https://d.domain.com' origin to access an endpoint within a service. However, before processing the request, the middleware responsible for validating Cross-Origin Resource Sharing (CORS) policies intervened. After inspecting the request, CORS middleware determined that it did not comply with the defined policies and, as a result, rejected the request.

"Access to XMLHttpRequest at 'https://a-push.domain.com/notifyhub/negotiate' from origin 'https://d.domain.com' has been blocked by CORS policy: Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource."



Middleware Short-Circuiting

Short-circuiting is the process of stopping the execution of the middleware components in the pipeline and skipping the rest of them based on a specific condition. This is often done to improve performance or handle a particular situation without going through the middleware pipeline completely. For example, you might have a middleware component that checks for a specific condition and, if the condition is met, immediately returns a response. In this example, the middleware handles a request and generates a response without calling the other middleware components.



Precaution

Short-circuitry should be applied with caution as it may interfere with the execution sequence and the intended behavior of the application. It is essential to always take into account the particular requirements and potential adverse effects when making a decision to interrupt a middleware pipeline.

Middleware Configuration: Understanding the Map, Use, and Run Methods

ASP.NET Core middleware provides a flexible way to handle requests and responses in the application pipeline. Three key methods, Map, Use and Run, play a crucial role in configuring middleware components. Let's explore each of these methods and understand how they contribute to the request processing pipeline.

Map Method

The Map method is used to conditionally branch the middleware pipeline based on the request's path. It allows you to specify a branch of middleware to handle requests that match a certain path prefix.

```
app.Map("/path", builder =>
{
    builder.UseMiddleware1();
    builder.UseMiddleware2();
    // Additional middleware for the specified path
});
```

In this example, if the request path starts with "/path," the middleware specified inside the Map block will be executed.

Use Method

The Use method is a fundamental building block for adding middleware components to the pipeline. It is used to add middleware that executes for all requests or for a specific path.

```
app.Use(async (context, next) =>
{
    // Code to be executed before calling the next middleware
    await next();
    // Code to be executed after the next middleware has completed
});
```

Middleware added using the Use method is executed in the order it is registered in the pipeline. It's crucial for tasks such as logging, authentication, or any other processing that needs to occur for each request.

Run Method

The Run method is used to create terminal middleware that terminates the request pipeline. It does not call the next middleware in the pipeline, making it the last piece of middleware to run.

```
app.Run(async context =>
{
    // Handle the request and generate a response
});
```

The Run method is often used for tasks that should be the final action in the pipeline, such as returning a response directly without invoking additional middleware.

In ASP.NET Core, the Map, Use, and Run methods provide a powerful mechanism for configuring middleware and controlling the request processing pipeline. By understanding how these methods work, we can create a well-structured and efficient pipeline to handle various aspects of request and response processing in their applications.

6

Custom Middleware

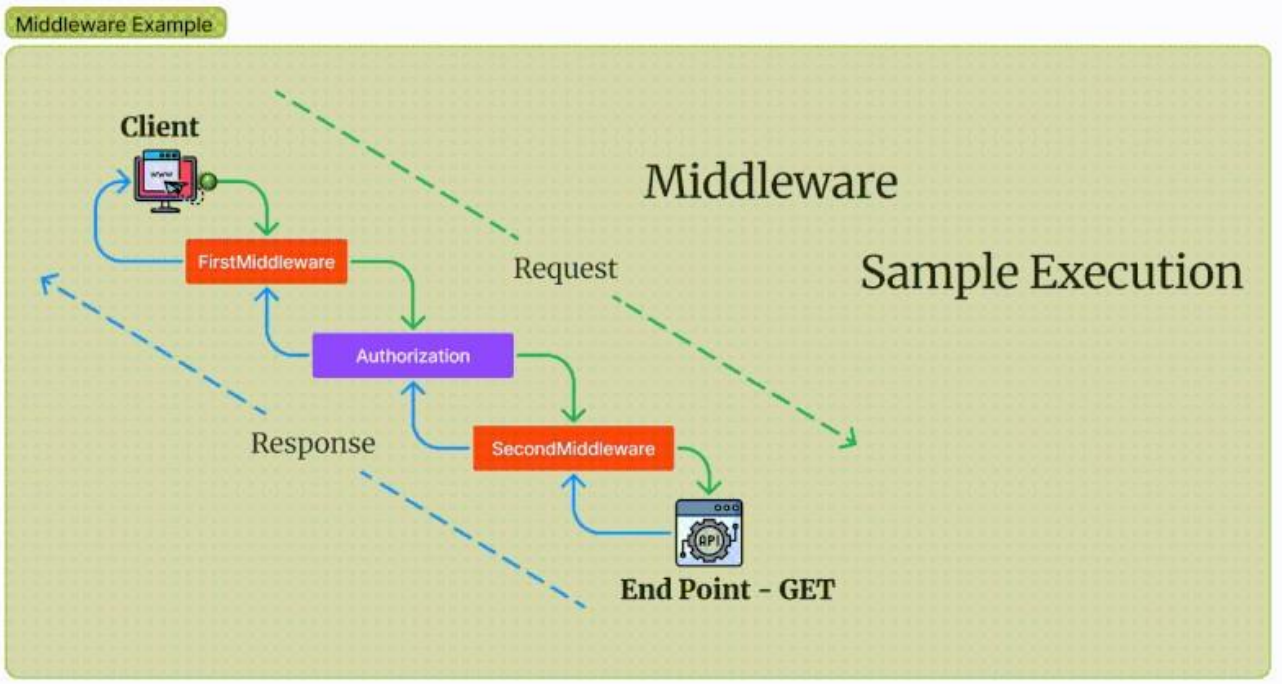
Overview

In this chapter, we will explore custom middleware in ASP.NET Core, offering a versatile solution for developers to execute various tasks such as authentication, logging, and error handling. It emphasizes modularity, reusability, and short-circuiting techniques for efficient request processing.

What is Custom Middleware?

Custom middleware serves as a robust and adaptable tool, empowering developers to execute a wide number of tasks, including but not limited to authentication, logging, and error handling. Its strength lies in its modularity and reusability, providing a flexible mechanism to seamlessly integrate custom logic into the request processing pipeline of ASP.NET Core applications. This versatility enables developers to craft tailored solutions that meet the specific needs of their projects while maintaining a modular and reusable codebase.

```
var app = builder.Build();

app.UseMiddleware<FirstMiddleware>(Guid.NewGuid());
app.UseAuthorization();
app.UseMiddleware<SecondMiddleware>(Guid.NewGuid());

//
// MapControllers is called to map attribute routed controllers.
// In the following example: HomeController matches a set of URLs similar
to what the default
// conventional route {controller=Home}/{action=Index}/{id?} matches
//
app.MapControllers();
app.Run();

//
// The following UseMiddleware<LastMiddleware> will never be invoked
because it's scheduled to run after
// the app.Run() method. The rule dictates that middleware must be added
before calling app.Run().
//
// app.UseMiddleware<LastMiddleware>(Guid.NewGuid()); // This line should
be removed or placed before app.Run()
```




```
FirstMiddleware
*****************
Entry. ID-> 7de251c04f3446c59ed27f981c838a3c 30/11/2023 04:39:06.041

    |-----------------
    |SecondMiddleware
    |-----------------
    |Entry. ID-> 449b6ca954174aec85d9b0324a29c417 30/11/2023 04:39:07.068

             => Entry. Hi, I'm Endpoint my details are below,
             MiddlewareCollections.Controllers.WeatherForecastController - `Get`
             Time: 30/11/2023 04:39:08.093

    |Exit. ID-> 449b6ca954174aec85d9b0324a29c417 30/11/2023 04:39:08.110
    |-----------------

Exit. ID-> 7de251c04f3446c59ed27f981c838a3c 30/11/2023 04:39:08.114
```

```
public static void Main(string[] args)
{
    var builder = WebApplication.CreateBuilder(args);

    // Add services to the container.

    builder.Services.AddControllers();
    // Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbu
    builder.Services.AddEndpointsApiExplorer();
    builder.Services.AddSwaggerGen();

    var app = builder.Build();

    // Register in the pipeline before calling Run()
    app.UseMiddleware<FirstMiddleware>(Guid.NewGuid());

    app.UseAuthorization();

    // Register in the pipeline before calling Run()
    app.UseMiddleware<SecondMiddleware>(Guid.NewGuid());

    ///
    /// MapControllers is called to map attribute routed controllers.
    /// In the following example: HomeController matches a set of URLs similar to what
    /// conventional route {controller=Home}/{action=Index}/{id?} matches
    ///
    app.MapControllers();

    app.Run();
    ///
    /// The function won't be invoked because it's scheduled to run after the app.Run()
    /// The rule dictates that middleware must be added before calling app.Run().
    /// !!! Not registered in the pipeline
    ///
    app.UseMiddleware<LastMiddleware>(Guid.NewGuid());
}
```

```
/// !!! Must Need  of this RequestDelegate  DI
// Inject RequestDelegate ->  for process an HTTP request.
public class FirstMiddleware(RequestDelegate next,Guid guid)
{
    private readonly RequestDelegate _next = next;

    /// !!! Must Need
    public async Task Invoke(HttpContext context)
    {
        Debug.WriteLine(" FirstMiddleware ");
        Debug.WriteLine("*****************");

        Debug.WriteLine($"Entry. ID-> {guid:n} {DateTime.UtcNow:dd/MM/yyyy hh:mm:ss.fff}

        Thread.Sleep(1000);

        // Call the next middleware in the pipeline
        await _next(context);

        Debug.WriteLine($"Exit. ID-> {guid:n} {DateTime.UtcNow:dd/MM/yyyy hh:mm:ss.fff}");
    }
}
```

- FirstMiddleware is a class that implements the middleware logic. It takes a RequestDelegate in its constructor, representing the next middleware in the pipeline.
- The InvokeAsync method is where you put your custom logic before and after calling the next middleware.
- app.UseMiddleware<FirstMiddleware>(Guid.NewGuid()); It adds your custom middleware to the application's request pipeline.

Furthermore, we will shed light on the concept of short-circuiting within the middleware architecture. Short-circuiting is a powerful technique that allows us to interrupt the usual flow of middleware execution, providing an early response under specific conditions. This can be instrumental in optimizing performance, implementing specialized handling, or even enforcing certain policies.

7

What is Short-Circuiting?

Overview

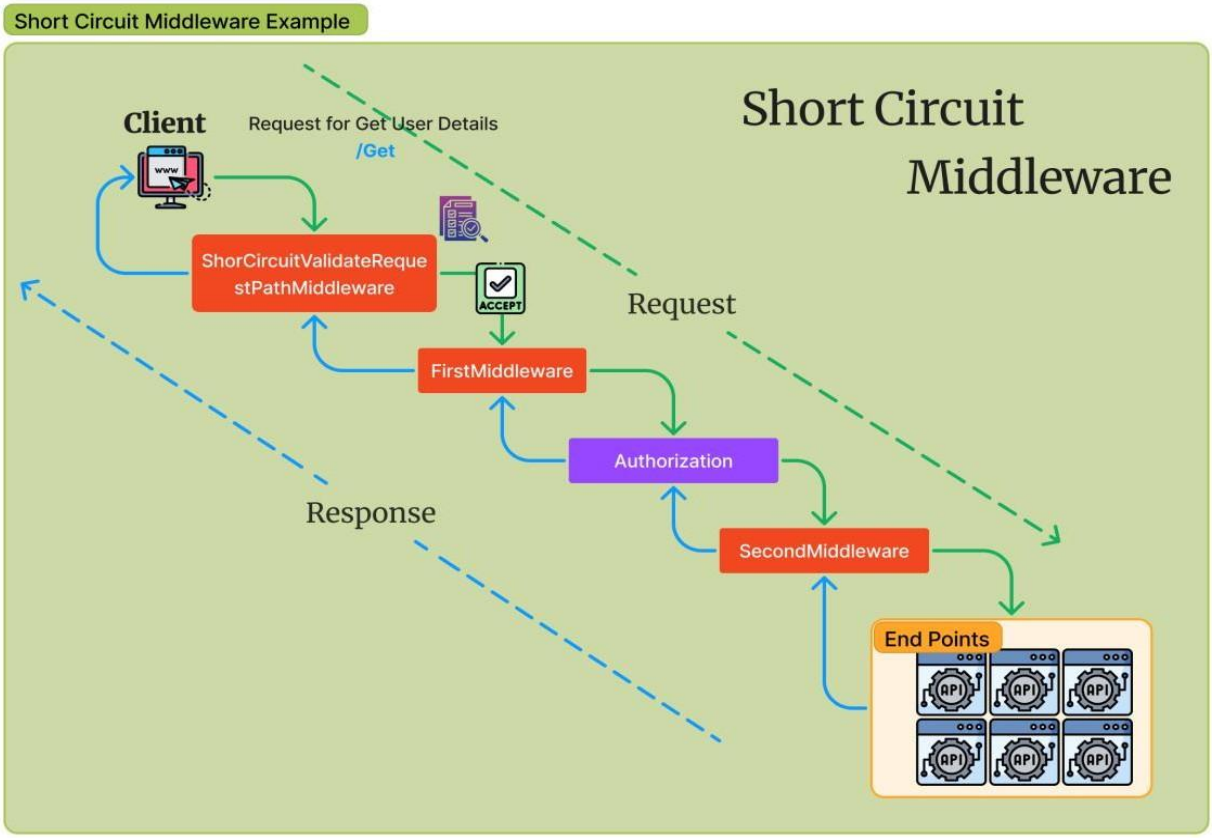
In this chapter, we will explore the intricacies of generic constraints and provides valuable insights into their applications and limitations. This chapter delves deep into the various constraints available in C# generics, enabling readers to make informed decisions when choosing appropriate constraints. It emphasizes the importance of striking a balance between flexibility and restriction and offers practical guidelines for designing generic classes and methods. With the inclusion of code examples and best practices, this chapter empowers developers to leverage the potential of generics in their .NET projects fully.

What is Short-Circuiting?

Short-circuiting is a powerful technique that allows us to interrupt the usual flow of middleware execution, providing an early response under specific conditions. This can be instrumental in optimizing performance, implementing specialized handling, or even enforcing certain policies.

```
public class ShortCircuitValidateRequestPathMiddleware
{
    private readonly RequestDelegate _next;

    private readonly IActionDescriptorCollectionProvider _actionDescriptorCollectionProvider;

    private readonly List<string?> path ;

    public ShortCircuitValidateRequestPathMiddleware(RequestDelegate next, IActionDescriptorCollectionProvider actionDescriptorCollectionProvider)
    {
        _next= next;
        _actionDescriptorCollectionProvider= actionDescriptorCollectionProvider;

        ///
        /// !!! For Example only, We need to lookup for better way.
        ///
        path = [];
        path = _actionDescriptorCollectionProvider.ActionDescriptors.Items.Where(
          ad => ad.AttributeRouteInfo != null).Select(
              ad => ad.AttributeRouteInfo?.Template).ToList();

        Debug.WriteLine("ShortCircuitValidateRequestPathMiddleware CTOR Called .....");
    }

    public async Task Invoke(HttpContext context)
    {
        Debug.WriteLine("ShorCircuit ");

        if (context.Request.Path.HasValue && path.Any(x => !string.IsNullOrWhiteSpace(x) &&
                        x.ToLower().Equals(context.Request.Path.Value.ToLower().Replace("/", ""))))
        {
            // Call the next middleware in the pipeline
            await _next(context);

            Debug.WriteLine($"\t Next Middleware Executed");
        }
        else {
            Debug.WriteLine($"\t Bye pass for unknow call.");
        }

    }
}
```

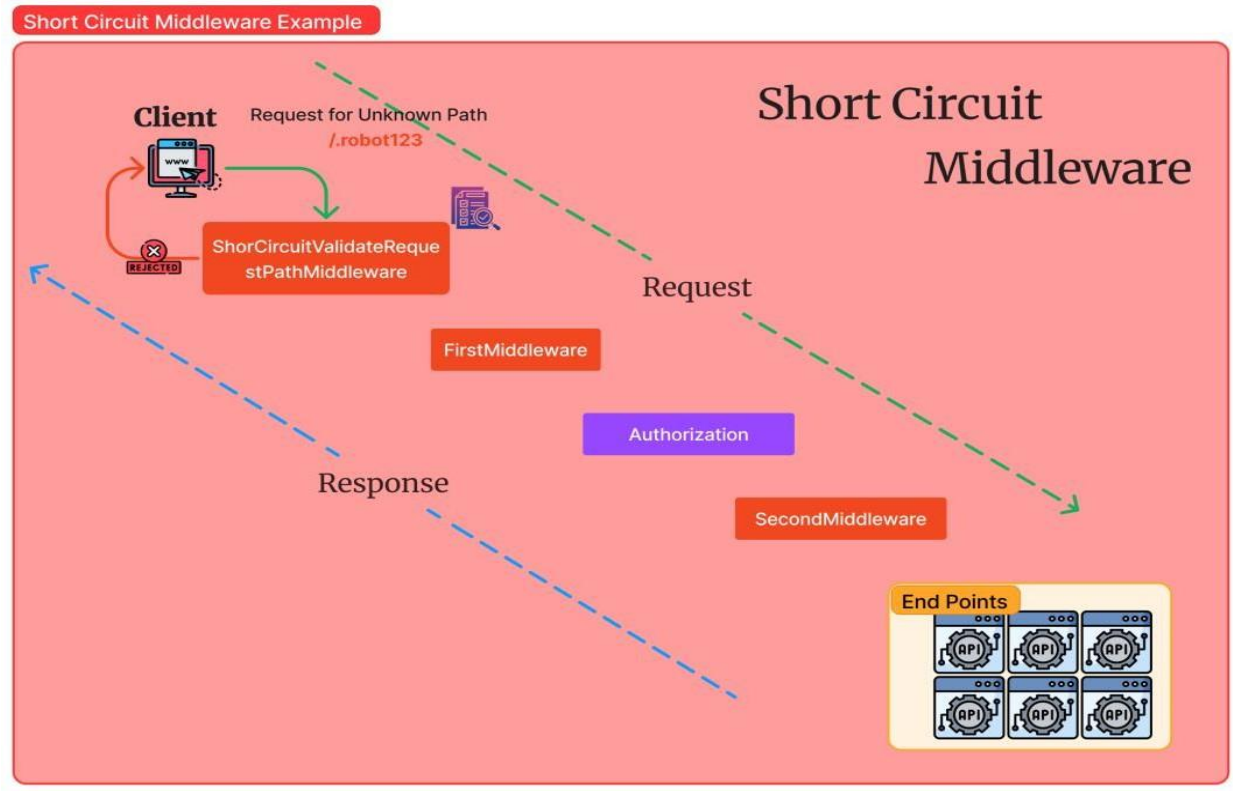
In the provided code snippet, during the initialization of the application, we gather and compile a comprehensive list of all potential routes. These route paths are then stored in an array. Upon receiving each incoming request, the service undergoes a check to verify whether the requested path exists within the locally stored array of routes. If the path is present, the request is permitted to proceed; otherwise, the service rejects the request, terminating its processing.

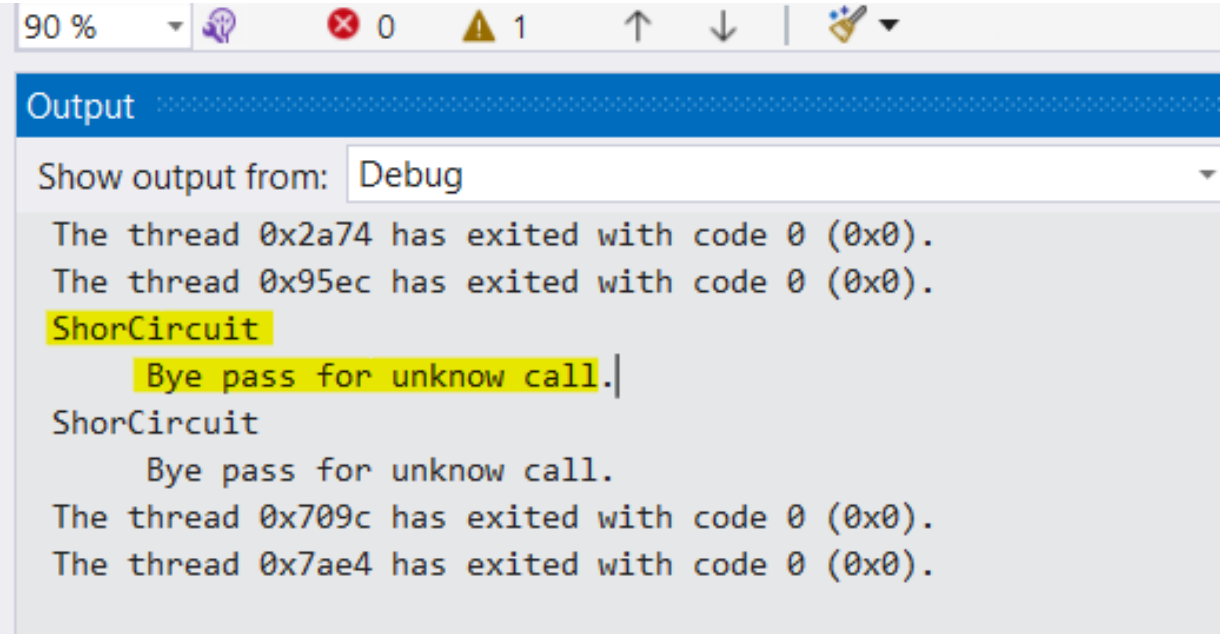



```
ShorCircuit
 FirstMiddleware
*****************
Entry. ID-> efee6ec60b5441259ece66477c131150 01/12/2023 09:48:21.427
    |-----------------
    |SecondMiddleware
    |-----------------
    |Entry. ID-> b60da17d78864eea979e75481cc6ae50 01/12/2023 09:48:22.444
'MiddlewareCollections.exe' (CoreCLR: clrhost): Loaded 'Anonymously Hosted DynamicMethods Assembly'.

             => Entry. Hi, I'm Endpoint my details are below,
             MiddlewareCollections.Controllers.WeatherForecastController - `Get`
             Time: 01/12/2023 09:48:23.479

    |Exit. ID-> b60da17d78864eea979e75481cc6ae50 01/12/2023 09:48:23.512
    |-----------------
Exit. ID-> efee6ec60b5441259ece66477c131150 01/12/2023 09:48:23.516
     Next Middleware Executed
```

Note:

- Request - /Get
- Short-Circuit Middleware verify the request path, /Get is Known one.
- So, it will process the request further and give the response. Check the Output Image in above.




90 %
0
1
Output
Show output from: Debug
The thread 0x2a74 has exited with code 0 (0x0).
The thread 0x95ec has exited with code 0 (0x0).
ShorCircuit
Bye pass for unknow call.
ShorCircuit
Bye pass for unknow call.
The thread 0x709c has exited with code 0 (0x0).
The thread 0x7ae4 has exited with code 0 (0x0).

Note:

- Request - /.robot123
- Short-Circuit Middleware verify the request path, /.robot123 is unknown path.
- So, it will reject the request and terminate the pipeline. Check the Output Image in above.

Short-circuiting in ASP.NET Core middleware, while powerful, comes with its own set of pros and cons. Let's explore both sides:

Pros:

Performance Optimization:

- Short-circuiting allows you to skip unnecessary middleware and immediately respond to a request based on certain conditions. This can significantly improve the performance of your application by avoiding the overhead of processing unnecessary middleware components.

Early Response:

- It enables you to send a response early in the request pipeline, which can be beneficial for scenarios where you want to handle requests without going through the entire pipeline.

Conditional Execution:

- Short-circuiting can be conditionally applied based on specific requirements, allowing for fine-grained control over when to interrupt the middleware pipeline.

Efficient Error Handling:

- It can be used for efficient error handling, enabling you to intercept and handle errors early in the pipeline before they propagate further.

Cons:

Loss of Middleware Functionality:

- Short-circuiting might result in skipping important middleware components that provide essential functionality. This could lead to unintended consequences if not carefully managed.

Debugging Complexity:

- Understanding the flow of execution becomes more complex when short-circuiting is employed. Debugging may become challenging, especially when different conditions trigger short-circuiting in various parts of the application.

Maintenance Challenges:

- As the number of conditions and short-circuiting scenarios increases, maintaining and updating the code may become challenging, potentially leading to code that is harder to understand and modify.

Ordering Dependency:

- The order of middleware registration becomes crucial when short-circuiting is employed. Dependencies and execution order need careful consideration to avoid unexpected behavior.

In summary

short-circuiting can be a valuable tool for optimizing performance and handling specific scenarios efficiently. However, it should be used thoughtfully, considering the potential trade-offs and impact on code maintainability. Striking the right balance ensures that short-circuiting enhances rather than hinders the overall functionality and maintainability of your ASP.NET Core application.

C#Corner

OUR MISSION

Free Education is Our Basic Need! Our mission is to empower millions of developers worldwide by providing the latest unbiased news, advice, and tools for learning, sharing, and career growth. We're passionate about nurturing the next young generation and help them not only to become great programmers, but also exceptional human beings.

ABOUT US

CSharp Inc, headquartered in Philadelphia, PA, is an online global community of software developers. C# Corner served 29.4 million visitors in year 2022. We publish the latest news and articles on cutting-edge software development topics. Developers share their knowledge and connect via content, forums, and chapters. Thousands of members benefit from our monthly events, webinars, and conferences. All conferences are managed under Global Tech Conferences, a CSharp Inc sister company. We also provide tools for career growth such as career advice, resume writing, training, certifications, books and white-papers, and videos. We also connect developers with their potential employers via our Job board. Visit C# Corner

MORE BOOKS